LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-625732

Optimizing Time Integration of Chemical-Kinetic Networks for Speed and Accuracy

R. A. Whitesides, M. J. McNenly, D. L. Flowers

March 8, 2013

8th U.S. National Combustion Meeting
Park City, UT, United States
May 19, 2013 through May 22, 2013

Disclaimer

8th U. S. National Combustion Meeting
Organized by the Western States Section of the Combustion Institute
and hosted by the University of Utah
May 19-22, 2013

Optimizing Time Integration of Chemical-Kinetic Networks for Speed and Accuracy

Russell A. Whitesides, Matthew J. McNenly, and Daniel L. Flowers

Computational Engineering Division, Lawrence Livermore National Laboratory, 7000 East Ave. Livermore, CA 94550

This paper describes enhancements to a sparse, adaptively-preconditioned, solver for chemical-kinetic ignition simulations. A concurrent submission [1] describes the adaptively preconditioned solver, showing orders of magnitude reduction in computational time for ignition delay calculations with large chemical mechanisms. Building on that solver implementation, this paper describes further improvements in three categories: 1) Fast vector exponentiation implemented in the Arrhenius and equilibrium rate constant calculations, 2) Fill-reducing matrix permutations for Jacobian factorization and back substitution, and 3) custom-code generation tailored to specific chemical mechanisms.

A series of vector exponentials developed by other authors are implemented in the code and tested for speed and accuracy. The accuracy of the functions is tested against the standard library exponential function and for their affect on the accuracy of calculated ignition delay times. The fastest, high precision exponential is found to be from the Intel MKL library and provides approximately a nine times speed-up over the standard exponential on the system tested. Alternatively, an implementation with a permissive distribution license gives approximately an eight times speedup with acceptable accuracy.

Fill-reducing matrix permutations are found to be an important factor governing matrix operation costs during time integration. We tested the COLAMD, MMD, Metis nested-dissection, and Reverse Cuthill-McKee methods. The SuperLU implementation of MMD was found to give the best permutations (minimizing matrix fill-in) for the chemical systems we investigated. Disabling diagonal pivoting in the permutation procedure further reduces the amount of fill-in.

The custom-code generator is designed to create tailored source files for the calculation of rate constants and species production rates for chemical mechanisms. This allows an optimizing compiler to produce the most optimal machine code for a mechanism by reordering operations to increase cache locality and reduces the number of operations necessary by completely unrolling loops.

Without these improvements the adaptively-preconditioned solver gives a 1600-fold reduction in computational time for ignition delay calculations with a 7172 species biodiesel mechanism (from 26 hours to under one minute). The improvements reported here provide an additional factor of two reduction in wall clock integration time for ignition delay calculations over a sweep of mechanism sizes from ten to 7172 species. For the 7172 species mechanism, ignition delay time is reduced to about 30 seconds, giving ~3000-fold speedup over the dense solver approach.

1. Introduction

Integration of reacting chemical networks is a computationally intensive problem important to research and development of many engineering problems including design of combustion devices, atmospheric chemistry modeling, and chemical reactor analysis. In a companion paper [1], a novel framework for fast integration of chemical networks is described that provides orders of magnitude reduction in wall clock time required for integration of large chemical mechanisms relative to previous approaches. This paper describes improvements to thermo-physical routines and integrator interface that further reduce the computational cost of time integration of chemical reactor networks.

2. Methods

CPU profiling of the ignition delay time calculations is used to determine the most computationally expensive areas of the simulations. Matrix LU factorization, LU back-substitution, and derivative, or right-hand-side (RHS), calculations are the three major areas of cost. From there, different strategies have been explored to reduce the costs in the identified computationally intensive areas. All calculations are performed on a dedicated compute node of a high-performance computing cluster with two Intel Xeon 5660 6-core processors per node. The test cases used to evaluate the improvements are ignition delay time calculations for a range of initial temperatures for ten different chemical mechanisms ranging in size from ten to 7172 species. This series of calculations is identical to those described in Section 3 of the companion paper [1]. Details of the major improvements to calculation speed are given in the following subsections.

Fast vector exponentiation.
The rate coefficients for chemical reactions are most commonly expressed in Arrhenius form,

$$k = AT^n e^{\frac{-E_a}{RT}}$$

The exponential term in such equations is the most expensive mathematical operation in evaluating the rates of change of species mass fractions in zero-dimensional reactor simulations. Lu and Long [2] suggested fast, approximate vector exponentiation functions to reduce the cost of RHS evaluations. The standard exponentiation function on most computing systems provides results that are accurately rounded to double precision representation. In most chemical simulations we can relax this requirement while still maintaining accurate simulation results. In addition, modern CPU implementations include single-instruction, multiple-data (SIMD) operations that allow a single operation to be performed simultaneously on two or more adjacent data [3]. With properly arranged data structures for the reaction rate coefficients, the rate equations can be calculated with SIMD operations.

We tested many vector exponential functions developed by others to compare their speed and the error in their approximation (relative to the standard exponential function):

- MKL [4]
- AMD LibM [5]
- Cephes [6]
- fmath [7]
- SLEEF [8]
- N. Okazaki [9]

The MKL and AMD LibM packages are closed source and the details of their implementations are not public. The other packages are open source and all of their implementations employ range reduction coupled with either polynomial fits or look-up tables. The Cephes package uses a Padé approximation with 7 terms. The fmath package uses an adjustable length look-up table. The SLEEF package employs a Taylor series approximation. The Okazaki code provides multiple functions with variable length Taylor and Remez approximations. The accuracy of all the available functions varies over the range of inputs. We characterized the accuracy using a max and average relative difference between each exponential method and results from the standard library [10] on 10^7 values spread evenly over the domain of the double precision exponential function (-708 to 708). The functions were then compared when used in the sweep of ignition delay calculations to see how the approximate values affected the calculated ignition delay time.

Fill-reducing matrix permutations.
The fast integrator relies on LU-decomposition and back-substitution of the Jacobian preconditioning matrix during the linear corrector step [11]. As described in the companion paper [1], a sparse matrix package (SuperLU [12-13]) is used to minimize the cost of these operations. A key aspect of the sparse algorithms is the inclusion of a permutation operation, which re-arranges the original matrix, A, to minimize the number of terms in the resulting decomposed matrices, L and U. The ratio of the number of non-zero terms in L and U to the number of non-zero terms in A is termed the fill ratio or fill factor. Reducing the fill ratio reduces the cost of each back-substitution by a similar amount. The problem of determining an optimal permutation is considered NP-complete. That is, one must essentially try all possible permutations to determine the optimal one. This quickly becomes intractable for matrices resulting from reactor simulations where the matrix size is on the order of the number of species included in the simulation. Heuristic methods have therefore been developed to yield close to optimal permutations with low computational cost. Reducing fill-in speeds up both the factorization and back-substitution calculations and so it is

desirable to explore as many options as possible. The SuperLU package includes some permutation algorithms and also allows for a user-generated permutation to be supplied. All of the SuperLU supplied algorithms (column approximate minimum degree (COLAMD), multiple-minimum degree on the structure of A transpose plus A (MMD_AT_PLUS_A), and multiple-minimum degree on the structure of A transpose times A (MMD_ATA)), as well as the Reverse-Cuthill-McKee [14], and METIS nested dissection [15] algorithms are tested. The goal is to balance permutation cost with the cost of LU decomposition and back substitution. As back substitution is necessary much more frequently than factorization, a moderately expensive permutation that gives significantly lower fill-ratios is favorable over a very inexpensive permutation that gives less optimal permutations. We also find that permutations can be saved from one factorization to the next while maintaining low fill-ratios if the preconditioner matrix is similar in structure to that from the previous factorization.

Custom-code generation.
In evaluating the chemical production rates for chemical reactors it is necessary to evaluate equations of the form

$$R_i = k_i \prod_j^{species} C_j^{\nu_{ij}}$$

and

$$\frac{dC_i}{dt} = \sum_j^{create} R_j - \sum_j^{destroy} R_j$$

These terms are evaluated in sparse form by maintaining lists of which species participate in each reaction. While this formulation minimizes the number of arithmetic steps necessary to compute the terms, it results in irregular data access patterns. When data are not read linearly from memory, the processor cache cannot be used as efficiently resulting in "lost" cycles while data is fetched from memory. In addition, the evaluation of these terms requires looping over all the reactants and products for the creation and destruction rates. The addition and comparison operations necessary to keep track of the looping variables can be a significant part of the total cost. An optimizing compiler can lessen this cost by "un-rolling" the loops and using longer strides, but some computational overhead will remain.

By forgoing generality and creating subroutines that explicitly define every operation for a particular chemical mechanism, both cache and loop inefficiencies can be minimized. The customized subroutines specify each reaction rate of progress, R_i, as the rate coefficient, k_i, multiplied by the concentration of each of that reaction's reactants, C_j, explicitly. In addition, each creation and destruction rate for each species is calculated through explicitly summing the rates of progress for the reactions that either create or destroy that species. Custom-code generating routines are written to create the necessary subroutines for specific mechanisms. Examples of the code used in the general case and custom code generated for a 10 species hydrogen mechanism [16] are given in Listings 1 and 2 of the Appendix. We also implemented code generation for the calculation of equilibrium constants for reversible reactions.

Preconditioner Matrix Update.
The fast chemical integrator employs the scaled-preconditioned generalized minimum residual (Spgmr) module of the CVODE integration package [1,11]. As part of the solver logic for this package, the user must supply a preconditioner setup routine, which calculates the Jacobian matrix, scales the result to form the preconditioning matrix, and performs matrix factorization. There are two modes for this update defined by the CVODE interface. In some cases, CVODE requests that all three steps are performed, otherwise only scaling of the previously calculated Jacobian matrix and matrix factorization are requested. In past versions of the chemical integrator, when dense matrix math was utilized, the matrix factorization accounted for 90% or more of the total integration cost for mechanisms of moderate to large size. As such, it was found that skipping the re-scaling and factorization when the preconditioner setup routine was called in the second mode yielded faster over-all integration times. With sparse-matrix algorithms and the improvements described above, this is no longer the case. The cost of additional factorizations is offset by a decrease in the number of integration steps, which now results in lower integration times.

3. Results and Discussion

Fast vector exponentiation.
The speed and accuracy of the various vector exponential functions compared to the exponential provided by the GNU C Library (glibc) are shown in Table 1. The relative time is calculated as the total wall clock time for 10^7 exponential evaluations (with input values evenly distributed from -708 to +708), divided by the time necessary for the same evaluations to be done using glibc [10]. Smaller relative time corresponds with faster calculations. The maximum and average relative errors are calculated over the same range of evaluations relative to values produced by glibc.

Exponential Package	Relative Time	Max Rel. Error	Avg. Rel. Error	Ref.
glibc	1.000	0.00E+00	0.00E+00	[10]
Cephes	0.527	3.14E-16	5.77E-17	[6]
Okazaki – Taylor-5	0.328	3.24E-06	7.18E-07	[9]
Okazaki – Taylor-7	0.427	7.03E-09	1.36E-09	[9]
Okazaki – Taylor-9	0.504	9.45E-12	1.64E-12	[9]
Okazaki – Taylor-11	0.591	8.79E-15	1.37E-15	[9]
Okazaki – Taylor-13	0.710	2.22E-16	4.99E-17	[9]
Okazaki – Remez-5	0.362	1.07E-07	5.54E-08	[9]
Okazaki – Remez-7	0.407	5.72E-11	2.97E-11	[9]
Okazaki – Remez-9	0.502	1.91E-14	9.91E-15	[9]
Okazaki – Remez-11	0.619	2.68E-16	6.17E-17	[9]
Okazaki – Remez-13	0.729	2.66E-16	6.12E-17	[9]
Okazaki – Remez-5S†	0.147	1.07E-07	5.54E-08	[9]
Okazaki – Remez-7S†	0.175	5.72E-11	2.97E-11	[9]
Okazaki – Remez-9S†	0.207	1.91E-14	9.91E-15	[9]
Okazaki – Remez-11S†	0.253	2.68E-16	6.17E-17	[9]
fmath – expd_v†	0.124	8.06E-14	2.44E-14	[7]
MKL (low accuracy)†	0.110	4.41E-16	1.07E-16	[4]
MKL (high accuracy)†	0.171	2.22E-16	1.18E-17	[4]
AMD LibM†	0.374	2.22E-16	1.12E-17	[5]
SLEEF†	0.451	2.22E-16	4.89E-17	[8]

Table1. Performance of fast, approximate exponential functions. †SIMD implementations.

For both speed and accuracy, MKL provides the best implementation, and gives some flexibility to trade some accuracy for faster results. If MKL is not an option, AMD LibM provides the best accuracy with fast results. The fmath package provides the fastest non-MKL results if its lower accuracy is acceptable for a given application. The nine and eleven term Remez routines from Okazaki fill in gaps in terms of speed/accuracy trade-off. The Taylor approximations are found to be uncompetitive due to lower accuracy than similar-cost Remez formulations. The Cephes and SLEEF packages are also not as attractive as other options due either to slow speed or low accuracy. With respect to the ignition delay calculations, the MKL package gives the shortest integration times and the fmath package provides similar speeds while being freely distributable. The lower accuracy of the fmath implementation is not found to significantly affect ignition delay time calculation accuracy. Computed ignition delay times for the sweep of mechanisms and initial temperatures are all within 0.01% of the results calculated using glibc for all of the tested exponential functions.

Fill-reducing matrix permutations.
The multiple-minimum degree algorithm on the structure of A transpose plus A (MMD_AT_PLUS_A) is found to provide the best trade-off of permutation cost versus fill-ratios for the chemical mechanisms examined. While the

METIS and Reverse-Cuthill McKee algorithms provide faster permutations, they are not found to decrease integration time due to higher fill-ratios compared to MMD_AT_PLUS_A. In addition, the SuperLU option to disable partial pivoting of the matrix yields lower fill-ratios while not causing stability issues.

Custom code generation.
The cost of calculating the well-stirred reactor right-hand-side is decreased by between 17 and 25% when using custom code for the species production rates and equilibrium constants. This decrease in cost translates directly to faster ignition delay time calculations.

Combined integrator speed-up.
The combined affects of the changes described above are shown in Figure 1 as decreases in the average integration times for all of the test cases. For mechanisms larger than hydrogen mechanism (10 species), the decrease in wall clock time is approximately a factor of two.

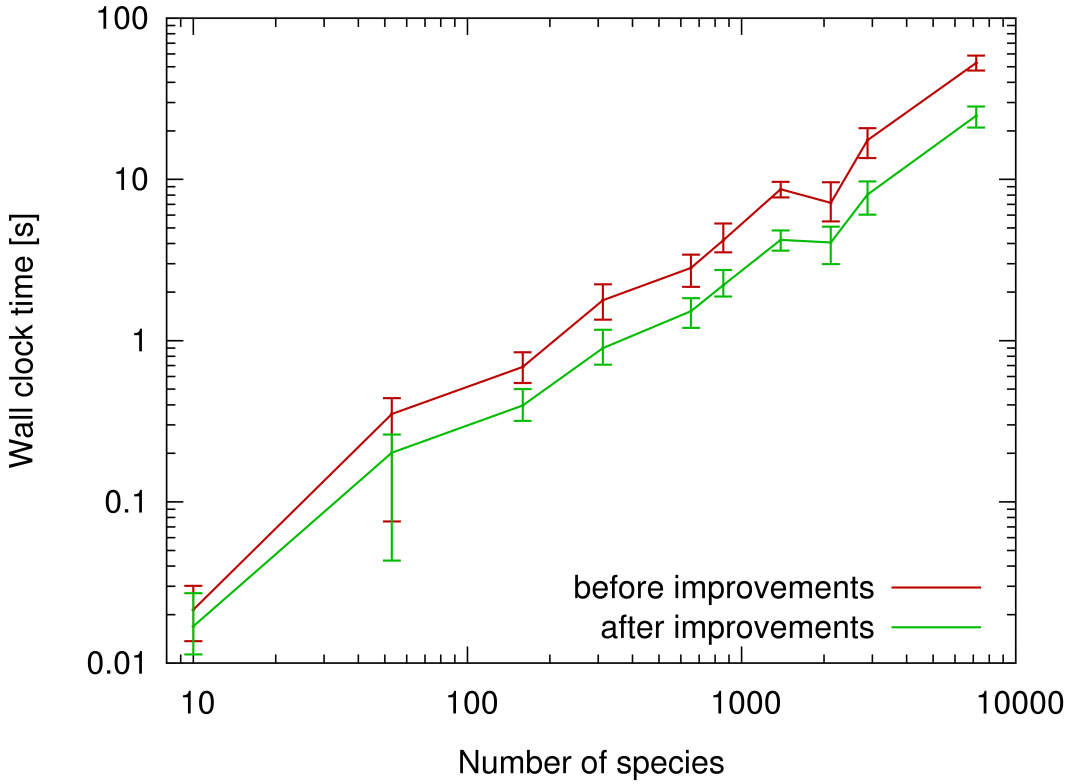

Figure 1. Average wall clock time for ignition delay time calculations as a function of chemical mechanism size before and after improvements to solver. Bars indicate minimum and maximum values of wall clock time for each mechanism temperature sweep.

The cost breakdowns for the major computational areas are shown in Figure 2. Decreases are seen in RHS evaluation, matrix permutation, and back-substitution. Fast exponentials and custom code generation speed up the RHS evaluation. Matrix permutation is improved by disabling partial pivoting and re-use of previous permutations. Back-substitution is improved due to decreased fill-in by disabling partial pivoting. The costs of Jacobian calculation and factorization are increased slightly due to more frequent evaluations, but these increased costs are counterbalanced by reduction in necessary integration steps, which helps to reduce the costs of all the other categories.

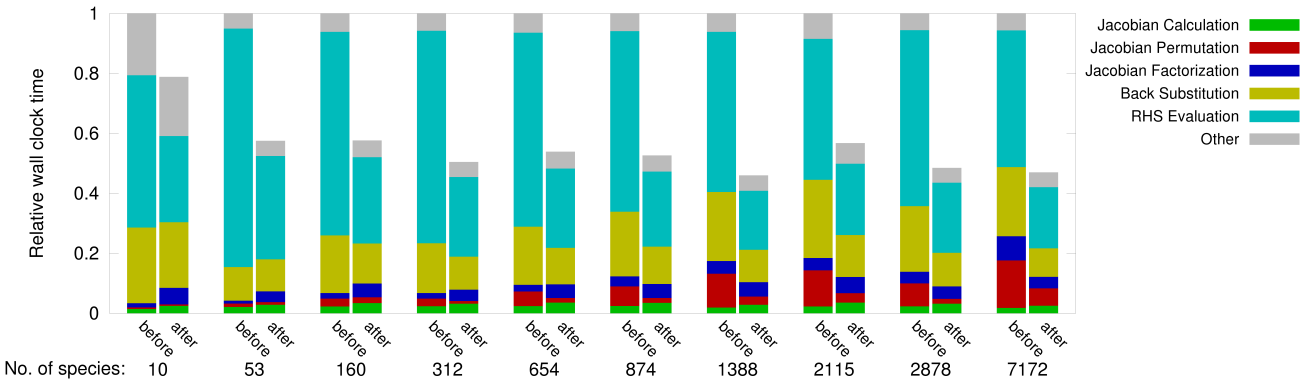

Figure 2. Break down of simulation costs in wall clock time relative to pre-improvement methodology.

4. Conclusions

We have identified and reduced the most computationally costly aspects of ignition delay calculations that remained after application of sparse, adaptively-preconditioned, iterative solver. The average ignition delay time calculation in our test cases is sped up by an additional factor of two over the original, orders-of-magnitude improvement. The breakdown of costs for our best case shows that simulation costs are fairly evenly distributed among different simulation tasks, meaning that further improvements and speed-ups in this framework will require significant simultaneous improvements to multiple elements of the solver. Alternate approaches to reduce integration time further include novel integrators able to take longer time steps while maintaining error control (e.g. exponential methods), error-bounded model reduction, and alternative compute hardware platforms such as GPUs.

Acknowledgements

This research was funded by DOE, Office of Vehicle Technologies, Gurpreet Singh, Combustion Team Leader. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] M. J. McNenly, R. A. Whitesides, and D. L. Flowers. Adaptive Preconditioning Strategies for Integrating Large Kinetic Mechanisms. In *8th US Nat Combust Meet*, No. 070RK-377, Park City, Utah, 2013.

[2] T. Lu and C. K. Law. *Prog Energy Combust Sci,* 35 (2009) 192–215.

[3] D. A. Patterson and J. L. Hennessey. *Computer Organization and Design: the Hardware/Software Interface*, 4th Edition, Morgan Kaufmann Publishers, Inc., San Francisco, 2012.

[4] Intel Corp., Intel Math Kernel Library 10.8.3, 2012. <http://software.intel.com/en-us/intel-mkl>

[5] AMD Inc., AMD LibM 3.0.2, 2012. <http://developer.amd.com/tools/cpu-development/libm/>

[6] S. Moshier, Cephes Mathematical Function Library 2.8, 2010. <http://www.moshier.net/#Cephes>

[7] S. Mitsunari, fmath: fast approximate function of float exp(float) and float log(float), 2012. <http://homepage1.nifty.com/herumi/soft/fmath.html>

[8] N. Shibata, SLEEF - SIMD Library for Evaluating Elementary Functions, 2013. <http://shibatch.sourceforge.net/>

[9] N. Okazaki, Fast exp(x) computation (with SSE2 optimizations), 2010. <http://www.chokkan.org/software/dist/fastexp.c.html>

[10] Free Software Foundation, Inc., GNU C Library version 2.12, 2010. <http://www.gnu.org/software/libc/>

[11] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. *ACM Trans Math* Software, 31 (2005) 363-396.

[12] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. *SIAM J Matrix Anal A*, 20 (1999) 720–755.

[13] X. S. Li et al. *SuperLU Users' Guide*. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. http://crd.lbl.gov/~xiaoye/SuperLU/. Last update: August 2011
[14] J. A. George and J. W-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.
[15] G. Karypis and V. Kumar. *SIAM J Sci Comput*, 20 (1999) 359—392.
[16] M. O'Connaire, H. J. Curran, J. M. Simmie, W. J. Pitz, and C. K. Westbrook. *Int J Chem Kinet*, 36 (2004) 603–622.

Appendix

Example code for custom-code generation.

Listing 1. Generic method.

```
calculateReactionCoeffs(T,speciesConcentrations,reactionRateOfProgress);

for(j=0; j<nTotalReactingSpecies; ++j)
 {reactionRateOfProgress[reactantStepIdxList[j]]*=speciesConcentrations[reactantSpcIdxList[j]];}

for(j=0; j<nSpecies; ++j)
{
  speciesCreationRate[j] = 0.0;
  speciesDestructionRate[j]=0.0;
}

for(j=0; j<nTotalReactingSpecies; ++j)
 {speciesDestructionRate[reactantSpcIdxList[j]]+=reactionRateOfProgress[reactantStepIdxList[j]];}

for(j=0; j<nTotalProducedSpecies; ++j)
 {speciesCreationRate[productSpcIdxList[j]]+=reactionRateOfProgress[productStepIdxList[j]];}
```

Listing 2. Custom method.

```
calculateReactionCoeffs(T,speciesConcentrations,reactionRateOfProgress);

reactionRateOfProgress[0]*=speciesConcentrations[0]*speciesConcentrations[3];
reactionRateOfProgress[1]*=speciesConcentrations[2]*speciesConcentrations[4];
reactionRateOfProgress[2]*=speciesConcentrations[1]*speciesConcentrations[2];
reactionRateOfProgress[3]*=speciesConcentrations[0]*speciesConcentrations[4];
reactionRateOfProgress[4]*=speciesConcentrations[1]*speciesConcentrations[4];
reactionRateOfProgress[5]*=speciesConcentrations[0]*speciesConcentrations[5];
reactionRateOfProgress[6]*=speciesConcentrations[2]*speciesConcentrations[5];
reactionRateOfProgress[7]*=speciesConcentrations[4]*speciesConcentrations[4];
reactionRateOfProgress[8]*=speciesConcentrations[1];
reactionRateOfProgress[9]*=speciesConcentrations[0]*speciesConcentrations[0];
reactionRateOfProgress[10]*=speciesConcentrations[3];
reactionRateOfProgress[11]*=speciesConcentrations[2]*speciesConcentrations[2];
reactionRateOfProgress[12]*=speciesConcentrations[4];
reactionRateOfProgress[13]*=speciesConcentrations[0]*speciesConcentrations[2];
reactionRateOfProgress[14]*=speciesConcentrations[5];
reactionRateOfProgress[15]*=speciesConcentrations[0]*speciesConcentrations[4];
reactionRateOfProgress[16]*=speciesConcentrations[0]*speciesConcentrations[3];
reactionRateOfProgress[17]*=speciesConcentrations[7];
reactionRateOfProgress[18]*=speciesConcentrations[0]*speciesConcentrations[7];
reactionRateOfProgress[19]*=speciesConcentrations[1]*speciesConcentrations[3];
reactionRateOfProgress[20]*=speciesConcentrations[0]*speciesConcentrations[7];
reactionRateOfProgress[21]*=speciesConcentrations[4]*speciesConcentrations[4];
reactionRateOfProgress[22]*=speciesConcentrations[2]*speciesConcentrations[7];
reactionRateOfProgress[23]*=speciesConcentrations[3]*speciesConcentrations[4];
reactionRateOfProgress[24]*=speciesConcentrations[4]*speciesConcentrations[7];
reactionRateOfProgress[25]*=speciesConcentrations[3]*speciesConcentrations[5];
reactionRateOfProgress[26]*=speciesConcentrations[3]*speciesConcentrations[8];
reactionRateOfProgress[27]*=speciesConcentrations[7]*speciesConcentrations[7];
reactionRateOfProgress[28]*=speciesConcentrations[3]*speciesConcentrations[8];
reactionRateOfProgress[29]*=speciesConcentrations[7]*speciesConcentrations[7];
reactionRateOfProgress[30]*=speciesConcentrations[8];
```

```
reactionRateOfProgress[31]*=speciesConcentrations[4]*speciesConcentrations[4];
reactionRateOfProgress[32]*=speciesConcentrations[0]*speciesConcentrations[8];
reactionRateOfProgress[33]*=speciesConcentrations[4]*speciesConcentrations[5];
reactionRateOfProgress[34]*=speciesConcentrations[0]*speciesConcentrations[8];
reactionRateOfProgress[35]*=speciesConcentrations[1]*speciesConcentrations[7];
reactionRateOfProgress[36]*=speciesConcentrations[2]*speciesConcentrations[8];
reactionRateOfProgress[37]*=speciesConcentrations[4]*speciesConcentrations[7];
reactionRateOfProgress[38]*=speciesConcentrations[4]*speciesConcentrations[8];
reactionRateOfProgress[39]*=speciesConcentrations[5]*speciesConcentrations[7];
reactionRateOfProgress[40]*=speciesConcentrations[4]*speciesConcentrations[8];
reactionRateOfProgress[41]*=speciesConcentrations[5]*speciesConcentrations[7];

speciesDestructionRate[0]=reactionRateOfProgress[0]+reactionRateOfProgress[3]
+reactionRateOfProgress[5]+reactionRateOfProgress[9]+reactionRateOfProgress[9]
+reactionRateOfProgress[13]+reactionRateOfProgress[15]+reactionRateOfProgress[16]
+reactionRateOfProgress[18]+reactionRateOfProgress[20]+reactionRateOfProgress[32]
+reactionRateOfProgress[34];

speciesDestructionRate[1]=reactionRateOfProgress[2]+reactionRateOfProgress[4]
+reactionRateOfProgress[8]+reactionRateOfProgress[19]+reactionRateOfProgress[35];

speciesDestructionRate[2]=reactionRateOfProgress[1]+reactionRateOfProgress[2]
+reactionRateOfProgress[6]+reactionRateOfProgress[11]+reactionRateOfProgress[11]
+reactionRateOfProgress[13]+reactionRateOfProgress[22]+reactionRateOfProgress[36];

speciesDestructionRate[3]=reactionRateOfProgress[0]+reactionRateOfProgress[10]
+reactionRateOfProgress[16]+reactionRateOfProgress[19]+reactionRateOfProgress[23]
+reactionRateOfProgress[25]+reactionRateOfProgress[26]+reactionRateOfProgress[28];

speciesDestructionRate[4]=reactionRateOfProgress[1]+reactionRateOfProgress[3]
+reactionRateOfProgress[4]+reactionRateOfProgress[7]+reactionRateOfProgress[7]
+reactionRateOfProgress[12]+reactionRateOfProgress[15]+reactionRateOfProgress[21]
+reactionRateOfProgress[21]+reactionRateOfProgress[23]+reactionRateOfProgress[24]
+reactionRateOfProgress[31]+reactionRateOfProgress[31]+reactionRateOfProgress[33]
+reactionRateOfProgress[37]+reactionRateOfProgress[38]+reactionRateOfProgress[40];

speciesDestructionRate[5]=reactionRateOfProgress[5]+reactionRateOfProgress[6]
+reactionRateOfProgress[14]+reactionRateOfProgress[25]+reactionRateOfProgress[33]
+reactionRateOfProgress[39]+reactionRateOfProgress[41];

speciesDestructionRate[6]=0.0;

speciesDestructionRate[7]=reactionRateOfProgress[17]+reactionRateOfProgress[18]
+reactionRateOfProgress[20]+reactionRateOfProgress[22]+reactionRateOfProgress[24]
+reactionRateOfProgress[27]+reactionRateOfProgress[27]+reactionRateOfProgress[29]
+reactionRateOfProgress[29]+reactionRateOfProgress[35]+reactionRateOfProgress[37]
+reactionRateOfProgress[39]+reactionRateOfProgress[41];

speciesDestructionRate[8]=reactionRateOfProgress[26]+reactionRateOfProgress[28]
+reactionRateOfProgress[30]+reactionRateOfProgress[32]+reactionRateOfProgress[34]
+reactionRateOfProgress[36]+reactionRateOfProgress[38]+reactionRateOfProgress[40];

speciesDestructionRate[9]=0.0;


speciesCreationRate[0]=reactionRateOfProgress[1]+reactionRateOfProgress[2]
+reactionRateOfProgress[4]+reactionRateOfProgress[8]+reactionRateOfProgress[8]
+reactionRateOfProgress[12]+reactionRateOfProgress[14]+reactionRateOfProgress[17]
+reactionRateOfProgress[19]+reactionRateOfProgress[21]+reactionRateOfProgress[33]
+reactionRateOfProgress[35];

speciesCreationRate[1]=reactionRateOfProgress[3]+reactionRateOfProgress[5]
+reactionRateOfProgress[9]+reactionRateOfProgress[18]+reactionRateOfProgress[34];

speciesCreationRate[2]=reactionRateOfProgress[0]+reactionRateOfProgress[3]
+reactionRateOfProgress[7]+reactionRateOfProgress[10]+reactionRateOfProgress[10]
+reactionRateOfProgress[12]+reactionRateOfProgress[23]+reactionRateOfProgress[37];

speciesCreationRate[3]=reactionRateOfProgress[1]+reactionRateOfProgress[11]
```

```
+reactionRateOfProgress[17]+reactionRateOfProgress[18]+reactionRateOfProgress[22]
+reactionRateOfProgress[24]+reactionRateOfProgress[27]+reactionRateOfProgress[29];

speciesCreationRate[4]=reactionRateOfProgress[0]+reactionRateOfProgress[2]
+reactionRateOfProgress[5]+reactionRateOfProgress[6]+reactionRateOfProgress[6]
+reactionRateOfProgress[13]+reactionRateOfProgress[14]+reactionRateOfProgress[20]
+reactionRateOfProgress[20]+reactionRateOfProgress[22]+reactionRateOfProgress[25]
+reactionRateOfProgress[30]+reactionRateOfProgress[30]+reactionRateOfProgress[32]
+reactionRateOfProgress[36]+reactionRateOfProgress[39]+reactionRateOfProgress[41];

speciesCreationRate[5]=reactionRateOfProgress[4]+reactionRateOfProgress[7]
+reactionRateOfProgress[15]+reactionRateOfProgress[24]+reactionRateOfProgress[32]
+reactionRateOfProgress[38]+reactionRateOfProgress[40];

speciesCreationRate[6]=0.0;

speciesCreationRate[7]=reactionRateOfProgress[16]+reactionRateOfProgress[19]
+reactionRateOfProgress[21]+reactionRateOfProgress[23]+reactionRateOfProgress[25]
+reactionRateOfProgress[26]+reactionRateOfProgress[26]+reactionRateOfProgress[28]
+reactionRateOfProgress[28]+reactionRateOfProgress[34]+reactionRateOfProgress[36]
+reactionRateOfProgress[38]+reactionRateOfProgress[40];

speciesCreationRate[8]=reactionRateOfProgress[27]+reactionRateOfProgress[29]
+reactionRateOfProgress[31]+reactionRateOfProgress[33]+reactionRateOfProgress[35]
+reactionRateOfProgress[37]+reactionRateOfProgress[39]+reactionRateOfProgress[41];

speciesCreationRate[9]=0.0;
```